Translating Task Models to State Machines

Andreas Wagner¹ and Christian Prehofer²

¹itestra GmbH, München, Germany
²fortiss GmbH, München, Germany

Keywords: Task Models, Statecharts, Partial State Machines, Model Transformation, Interaction Modeling.

Abstract: We present a new method to translate the established and well-known ConcurTaskTree (CTT) task modeling technique into state machines. For this purpose, we develop the concepts of partial state machines, Connectables and a connect operator, which form the theoretical framework for a new algorithm. For the translation, we develop and present a recursive, bottom-up algorithm, which exploits the inherent structure of CTTs and produces valid, partial state machines. This allows new development processes in the model-driven application and system development domain.

1 INTRODUCTION

Task models are a widely used technique to model user interaction with a system. They describe a set of possible arrangements of basic tasks, which can be executed to achieve the overall goal. ConcurTaskTrees (CTT) (Paternò, 2000) are one of the most well-known approaches to task modeling. Key features of CTTs include focus on activities, hierarchical structure, graphical syntax and a rich set of temporal operators (Paternò, 2001). The hierarchical structure of CTTs allows for the decomposition of complex tasks into more fine-grained child tasks. Relationships between tasks are defined via temporal operators between adjacent siblings. For the purpose of this work, we focus on the key CTT operators Enabling (>>), Choice ([[[]]]), Concurrent ([][]) and Disabling (><).

As an example, consider the (simplified) CTT of operating a TV as depicted in Figure 1. It consists of ten tasks, which implement the basic features of a TV like switching channels, adjusting the volume and putting the TV in standby mode. For instance, if the next channel should be selected, the user has to initiate the channel switch by pushing the “Chan+” button and the TV performs the necessary action. Further, channel switching and adjusting the volume should be possible to be done in parallel. By turning the TV off, all other tasks are interrupted.

In this work, we present a recursive algorithm to translate CTTs, based on their tree structure, into state machines. Contrarily to existing work on CTT-to-state-machine-translation (e.g. (Luyten and Clerckx, 2003) or (Van Den Bergh and Coninx, 2007)), our algorithm is a fully recursive, structure- and semantic-preserving approach to create executable state machines from CTTs. As the major technical contribution of this paper, we provide ways to manage and build intermediate state machines, which are incomplete or partial state machines and are composed in a recursive way. We develop and present the concepts of partial state machines and partial state machine containers.

One reason why the translation into state machines is challenging is that the actual input events are in the leaves of the CTT, but the CTT semantics is presented recursively in a top down fashion. Hence, a recursive algorithm must carefully compose the partial state machines and ensure at any time that the correct set of possible input events is maintained. This issue can be demonstrated in our TV example. The choice between the subtree “Increase Volume” (i.e. “Press Vol+ Button”) and “Decrease Volume” (i.e. “Press Vol- Button”) must be passed along the whole tree hierarchy, because both tasks may ultimately be used to as first interaction. A fully-recursive algorithm must consider this behaviour. While existing work in this area often relies on some precalculated information (so called “Enabled Task Sets”, e.g. (Luyten and Clerckx, 2003) or (Van Den Bergh and Coninx, 2007)), our algorithm maintains the correct input events on the fly.

Wagner, A. and Prehofer, C.
Translating Task Models to State Machines.
DOI: 10.5220/0005681702010208
In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016), pages 201-208
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved
2 STATECHARTS

In this work, we follow statecharts as defined in (Eshuis, 2009) and (Pnueli and Shalev, 1991):

Definition 1. A statechart $SC$ is a tuple $\tau = (S, T, E)$, where

- $S$ is a set of states
- $T$ is a set of transitions
- $E$ is the set of events that transitions

For a given set $S$, which represents the set of all states and $\mathcal{P}(S)$ being the powerset of $S$, the function $\text{substates} : S \rightarrow \mathcal{P}(S)$ maps a given state $s$ to the set of its immediate children.

States may have a property initial, which indicates if a given state is an initial state. The function $\text{isInitial} : S \rightarrow \{true, false\}$ assigns each state an indicator whether it is initial or not. Basic states are defined by means of the substates function. If the substates set is empty for a state $s \in S$, i.e. $\text{substates}(s) = \emptyset$, then $s$ is called a “basic state”. Basic states may have a final property (Omg, 2004). To check whether a basic state is final or not, we use the function $\text{isFinal} : S_h \rightarrow \{true, false\}$, which for an element of the set of all basic states $S_h$ indicates if the state is final or not.

Like basic states, composite states are defined by means of the substates function. If the substates set is not empty for a state $s \in S$, i.e. $\text{substates}(s) \neq \emptyset$, then $s$ is called a “composite state”. Composite states must be either of type AND or OR. If $S_c$ is the set of all composite states, the function $\text{type} : S_c \rightarrow \{\text{AND}, \text{OR}\}$ assigns for each composite state its type (Pnueli and Shalev, 1991).

OR-States indicate sequential behavior (Eshuis, 2009). An OR-State is defined by means of the substates function and the type function (Pnueli and Shalev, 1991). If a state $s \in S$ is of type OR and has substates, i.e. $\text{substates}(s) \neq \emptyset \land \text{type}(s) = \text{OR}$ it is called “OR-State”. The set of all OR-States is denoted $S_{OR}$. It especially holds that when the statechart is in state $s$, it is also in one and only one state $s' \in \text{substates}(s)$.

Throughout this work, we call OR-States compound or hierarchical states. Each compound state $s$ has an initial state $S_{init}$, which is an element of $\text{substates}(s)$. The initial state can be obtained by the function $\text{initial} : S_{OR} \rightarrow S$.

AND or parallel states indicate parallel behavior (Eshuis, 2009). Analogously to OR-States, AND-States are defined by means of the substates and type function (Pnueli and Shalev, 1991). If a state $s \in S$ is of type AND and has substates, i.e. $\text{substates}(s) \neq \emptyset \land \text{type}(s) = \text{AND}$ it is called “AND-State”.

When the statechart is in state $s$, it is simultaneously in all states $s' \in \text{substates}(s)$. Further, we require the substates $s' \in \text{substates}(s)$ to be composite states.

Like states, transitions have properties which can be obtained by corresponding functions. First, we denote the set of all transitions $T$ and introduce the functions source and target on it. Both source: $T \rightarrow S \cup \{\text{null}\}$ and target: $T \rightarrow S \cup \{\text{null}\}$ are partial functions which for a transition $t \in T$ either return a $s \in S$ or null if source($t$) respectively target($t$) is undefined.

More details on the employed functions and predicates for state machines can be found in (Wagner, 2015).

3 PARTIAL STATE MACHINES

Partial State Machines will be be used as intermediate results of the translation algorithm. To combine par-
tional state machines, we use the concept of Connectables and a connect operator, as defined below.

Connectables are basically well-defined interfaces for the composition of partial state machines. Connectables rely on free transitions, which we define first. A transition $t \in T$ is said to be free iff $\text{target}(t) = \text{null} \lor \text{source}(t) = \text{null}$. We denote the set of all free transitions $T_{\text{free}}$ and the complementary set of non-free transitions $T \setminus T_{\text{free}} = T_{\text{nonfree}}$.

With the definition of free transitions at hand, Connectables can be defined:

Definition 2. Let $S_B$ be the set of all basic states, $S_C$ be the set of all compound states and $S_P$ be the set of all parallel states. Further, let $T_{\text{free}}$ be the set of all free transitions. A Connectable $c$ is then defined as

$$ c \in (S_B \cup S_C \cup S_P \cup T_{\text{free}}) $$

Consequently, a Connectable is an abstraction over states and transitions. Often the actual type of a Connectable is needed. For this purpose, we use the function $\text{role}: C \mapsto R$, where $R = \{\text{Transition, BasicState, CompoundState, ParallelState}\}$ is the set of all roles and $C$ is the set of all Connectables.

Partial state machines (PSMs) are a superset of statecharts, which includes “incomplete” statecharts where e.g. transitions may not lead to a state. We define PSMs as follows:

Definition 3. A partial state machine is a tuple $\tau = (S, T, E, \delta)$ where

- $S \subset (S_B \cup S_C \cup S_P)$ is a set of states
- $T \subset (T_{\text{free}} \cup T_{\text{nonfree}})$ is a set of transitions
- $E$ is a set of events that trigger transitions
- $\delta: S \times T \rightarrow S$ is a partial transition function $S$ and $T$ may be empty, however the restriction holds that if $S = \emptyset, T \neq \emptyset$ and vice versa.

Figure 2 shows - in bold lines - some examples for (simple) partial state machines (the dotted lines are explained below).

Next we introduce in and out sets on PSMs. Both sets are in essence a set of the “dangling” components of the PSM. As in set $I$ we denote all free transitions of a PSM which don’t have a source state and all states which don’t have ingoing transitions. Consequently, the out set $O$ of a PSM consists of all free transitions which don’t have a target state and all states which don’t have outgoing transitions. Note that both $I$ and $O$ may contain basic, compound or parallel states. Further, we define that $I$ consists of only transitions or exactly one state. As an example consider the PSM in Figure 2(d). The in set consists of the two (free) transitions “EventA_x” and “EventA_y”, whereas the out set consists of the two (free) transitions “EventB_x” and “EventB_y”. A formal definition of in and out sets can be found in (Wagner, 2015).

Finally, we need a function $\text{states}: P \mapsto \mathcal{P}(S)$, which returns all the states (basic states, compound states, parallel states) of a given PSM. With the in sets, states set and out sets, we can define the concept of PSM containers:

Definition 4. Let $P$ be a partial state machine, $T_P$ the set of transitions of $P$ and $S_P$ the set of states of $P$. A partial state machine container $P_C$ is then a 3-tuple $\tau = (I_P, O_P, S_P)$ where

- $I_P$ is the in set of $P$ with $I_P \neq \emptyset$
- $O_P$ is the out set of $P$ with $O_P \neq \emptyset$
- $S_P$ is the set of states of $P$

Obviously, a PSM container is an abstraction over a PSM, which hides the internal structure and only exposes its interfaces (i.e. Connectables) to the outside world. Figure 2 shows - in dotted lines - the PSM containers for the PSMs in this figure. Note that the corresponding container of a PSM can be calculated at runtime by means of the in, out and states functions. Therefore, we will often use the terms partial state machine (PSM) and partial state machine container (PSMC) interchangeably throughout this work. This is only for convenience and does not have any implications on the translation algorithms.

Finally, the connect-operator (denoted by $\xrightarrow{\text{connect}}$) is a binary function which maps Connectables to sets of states and transitions. It takes two Connectables $c_1$ and $c_2$ as parameters and returns a set consisting of states and transitions with modified properties.
Most importantly, the \textit{connect} operator guarantees that the Connectables \(c_1\) and \(c_2\) are connected in a proper way, i.e. a state is followed by a transition and vice versa. Informally, its purpose is to check if two Connectables can be connected and to return the possible connections. We will use it to connect two Connectables of different partial state machines.

The \textit{connect} operator is formally defined as follows:

\textbf{Definition 5.} Let \(C\) be the set of all Connectables, \(S = S_{\text{free}} \cup S_{\text{conf}} \cup S_{\text{partial}}\) be the set of all states and \(T = T_{\text{free}} \cup T_{\text{conf}} \cup T_{\text{partial}}\) the set of all transitions. Further, let \(X = S \cup T \cup \mathcal{P}(X)\) the powerset of \(X\). Then the function \(\text{connect} : C \times C \rightarrow \mathcal{P}(X)\) is defined as

\[
\text{connect}(c_1, c_2) = \begin{cases} 
\mathcal{P}(X) & \text{if } c_1 = c_2 \\
\{ s \in X \mid s \text{ is a (free) transition and } s \text{ points to } c_2 \} & \text{if } c_1 \text{ is a (free) transition and } c_1 \text{ is connected to } c_2 \text{ by means of an intermediate basic state } \text{Intermediate}\text{; the target of } c_1 \text{ then points to } \text{Intermediate}\text{, the outgoingTransitions set of } \text{Intermediate}\text{ contains } c_2\text{, the ingoingTransitions set of } \text{Intermediate}\text{ contains } c_1\text{, and the source of } c_2 \text{ points to } \text{Intermediate}\} & \text{if } c_1 \text{ is connected to } c_2 \text{ by means of an intermediate transition } \text{Intermediate}\text{; The outgoingTransitions set of } c_1 \text{ and the outgoingTransitions set of } c_2 \text{ then contain } \text{Intermediate}\text{, the target of } \text{Intermediate}\text{ points to } c_2 \text{ and the source of } \text{Intermediate}\text{ points to } c_1\end{cases}
\]

For its behavior, four cases have to be distinguished. If a Connectable \(c_1\) is a (free) transition and a Connectable \(c_2\) is a (free) transition, then \(c_1\) is connected to \(c_2\) by means of an intermediate basic state \(\text{Intermediate}\). The target of \(c_1\) then points to \(\text{Intermediate}\), the outgoingTransitions set of \(\text{Intermediate}\) contains \(c_2\), the ingoingTransitions set of \(\text{Intermediate}\) contains \(c_1\), and the source of \(c_2\) points to \(\text{Intermediate}\).

If \(c_1\) is a (free) transition and \(c_2\) is a state, then \(c_1\) is directly connected to \(c_2\). The target of \(c_1\) then points to \(c_2\), and the ingoingTransitions set of \(c_2\) contains \(c_1\). The reverse case is handled analogously. Finally, if both \(c_1\) is a state and \(c_2\) is a state, then \(c_1\) is connected to \(c_2\) by means of an intermediate transition \(\text{Intermediate}\). The outgoingTransitions set of \(c_1\) and the outgoingTransitions set of \(c_2\) then contain \(\text{Intermediate}\), the target of \(\text{Intermediate}\) points to \(c_2\) and the source of \(\text{Intermediate}\) points to \(c_1\).

\section{4 TRANSLATION RULES}

The translation of CTTs into state machines is based on a recursive algorithm, where each recursive call returns a partial state machine. The algorithm is described in full detail (including proof-sketches employing induction) in (Wagner, 2015).

We start by explaining the basic case of the recursion, namely tasks. Tasks are the leafs of a CTT. Within an application, they perform the actual computations, enable the user to interact with the system, or provide feedback to the user and the environment. For the translation, we only cover interaction tasks and application tasks. The type of a task directly influences its counterpart in the resulting PSM.

Interaction tasks require some sort of interaction with the user and/or environment, thus being always bound to a specific event. Events move the system forward and may initiate an arbitrary computation or response from the system. In state machines, events are represented as transitions (e.g. (Harel, 1987)). Thus, we translate an interaction task into a \textbf{transition}.

On the other hand, it is statically known that an application task is performed on the device and does not require any interaction with the user. Because of this, an application task can be seen as an isolated computation unit, which is entered when the computation must be done and left when the computation finished. We therefore translate an application task into a \textbf{basic state}.

Besides the events which correspond to interaction tasks, additional events must be introduced which notify the system about the completion of application tasks (i.e. that states finished their execution). These events are called notification events and trigger \textit{notification transitions}, which are attached to the states which correspond to the application tasks.

As an additional step, the generated states and transitions must be wrapped into a partial state machine container \(\text{PSM}_{\text{interaction}}\) and \(\text{PSM}_{\text{Application}}\) respectively, which provide the Connectables for subsequent connection operations. For interaction tasks, the system is the only element created, it is simply returned as both \textit{in} and \textit{out} set of the PSM. For application tasks, the created state is returned as \textit{in} set and the notification transition is returned as \textit{out} set.

With the translation rules for the base case (i.e. tasks) at hand, we can now translate the CTT operators. In the following, we consider translations for the \textit{Enabling}, \textit{Choice}, \textit{Concurrent} and \textit{Disabling} operators. As other operators do not add essentially new concepts, we expect that the translation scheme can be extended to other operators analogously.

Note that CTT execution is in general non-deterministic, as there are choices to be taken in \textit{Choice}, \textit{Concurrent} and \textit{Disabling} operators. We will assume that these choices are essentially taken by external user interactions and not by random decisions. In case such non-determinism is desired, it can be modeled by external components which take the required decisions. For the purpose of specifying our assumption, we use the function \textit{first}, which is introduced in (Paternò, 2000). The informal purpose of this function is to return a set of tasks which should be executed first, given a CTT or a subtree. We require that the \textit{first} set for all \textit{Disabling}, \textit{Concurrent} and \textit{Choice} operators consists of only interaction tasks (i.e. tasks which require user interaction). Thus we can make sure that choices are always resolved externally (i.e. by the user).
4.1 Enabling Operator

According to the CTT specification provided in (Paternò et al., 2012), the semantics of Enabling requires the second task not to be activated until the first task is performed. This establishes a strictly sequential relation between the tasks of the operator. The partial state machine representing the Enabling operator must guarantee that this relation between $\alpha$ and $\beta$ is preserved during the execution phase (i.e. that $\alpha$ must be executed before $\beta$ in any case).

The actual translation of a CTT $\alpha >> \beta$ is shown in Algorithm 1. To seize on our example of Figure 1, we apply the algorithm to the Enabling operator with $\alpha$ being the task “Press Chan+ Button” and $\beta$ being the task “Go to next Channel”. First, the algorithm creates partial state machine containers $PSM_\alpha$ for $\alpha$ and $PSM_\beta$ for $\beta$ by applying the recursive translation function $translate$ to both tasks (or subtrees in general). The connection between both PSMs is done by connecting each component of $out(PSM_\alpha)$ (which is a transition representing the “press button” task) with each component of $in(PSM_\beta)$ (a state representing the “goto” task) by means of the $\rightarrow$ operator. Note that in general, $in(PSM_\beta)$ either contains only transitions or exactly one state. This is ensured by the definition of the $in$ set and the aforementioned requirements on the CTT.

The result of the connection operation is a new partial state machine $PSM_{Enabling}$ that contains transitions and states from $PSM_\alpha$ and $PSM_\beta$ as well as additional states introduced by the $\rightarrow$ operator (Figure 3). The $in$ set of $PSM_{Enabling}$ is the $in$ set of $PSM_\alpha$ and the $out$ set of $PSM_{Enabling}$ is the $out$ set of $PSM_\beta$. Note that for further connections, only the Connectables of $PSM_{Enabling}$ are relevant. The contents of the PSM container do not affect the translation of the upper operators.

4.2 Choice Operator

The Choice operator is described by Paternò to offer two tasks/subtrees and “once one has started the other one is no longer enabled” (Paternò et al., 2012). For our translation algorithm, we assume that choices always start with interaction tasks, as defined beforehand. Thus, we avoid any kind of possible non-determinism, because the choices are “externally” resolved by the user via explicit actions (e.g. input events).

The idea is to generate two, mutually exclusive PSMs, where each PSM can be entered via its own transition(s). The $in$ set of both state machines thus represents the “decision”, which subtree should be executed. Once $PSM_\alpha$ is active, there is no possibility to enter $PSM_\beta$ and vice versa.

The translation rule is shown by Algorithm 2. Considering our example, we translate the Choice between going to the next channel ($\alpha$) or to the previous channel ($\beta$). The main step is to generate partial state machines $PSM_\alpha$ and $PSM_\beta$ by applying the translation function $translate$ to both subtrees.

After translating the subtrees, we can merge both PSMs into a combined partial state machine $PSM_{Choice}$, which leads to a single PSM container representing the operator (Figure 4).

4.3 Concurrent Operator

The Concurrent operator allows independent and concurrent execution of both subtrees $\alpha$ and $\beta$. This
means, that at every step in time a task of $\alpha$ or $\beta$ can be executed without effecting each other. Paternò et al. define the tasks of the Concurrent operator to be “performed in any order, or at same time, including the possibility of starting a task before the other one has been completed” (Paternò et al., 2012). Consequently, all interleavings of the tasks of $\alpha$ and $\beta$ must be possible.

It is therefore feasible to translate the Concurrent operator directly into parallel states. Each subtree is represented by a substate of a common parallel state, which allows the corresponding tasks to be executed independently. The algorithm basically creates two parallel states - one with Connectables (transitions) for $\alpha$ and one with Connectables (transitions) for $\beta$. This is necessary in order to preserve the semantics of the Concurrent operator. If only one parallel state with both Connectables for $\alpha$ and $\beta$ would be created, the ingoing transitions would be mutually exclusive, which leads to a violation of the operator’s semantics.

The formal translation steps for Concurrent are shown by Algorithm 3. In our example, the sub-CTT “Operate TV” matches the translation rule with $\alpha$ being the subtree “Channel Selection” and $\beta$ being the subtree “Volume Selection”. For the translation, we first create corresponding partial state machines $PSM_\alpha$ and $PSM_\beta$ by applying translate to both subtrees. Next, $PSM_\alpha$ will be equipped with a new final state by connecting each transition of $PSM_\alpha$’s out set to it. This leads to a new partial state machine $PSM_{\alpha.final}$, $PSM_\beta$ (“Volume Selection”) on the other hand is attached with a new initial and final state. This turns the partial state machine into an ordinary executable state machine which we call $PSM_{\beta.inout}$. Note that $in(PSM_\beta) \neq in(PSM_{\beta.inout})$, because the transitions of $PSM_\alpha$’s in set were replaced by the common initial state.

Next, the ingoing transitions $transitions_\alpha$ of $PSM_{\alpha.final}$ are extracted and both $PSM_{\alpha.final}$ and $PSM_{\beta.inout}$ are wrapped into their own compound state $C_\alpha^1$ and $C_\beta^1$. Note that the ingoing transitions “Press Chan+ Button” and “Press Chan- Button” of $PSM_{\alpha.final}$ still point to their targets in $C_\alpha^1$. $C_\alpha^1$ and $C_\beta^1$ are then wrapped in a parallel state $P_1$. Finally, a completion transition $t_1^{completion}$ is created which is connected with $P_1$. The completion transition will be triggered automatically, if all of the parallel state’s substates reach their final state. The parallel state with Connectables for $\beta$ is created analogously, but with $PSM_\alpha$ and $PSM_\beta$ interchanged.

$PSM_{Concurrent}$ of the CTT “Operate TV” is shown in Figure 5.

Algorithm 3: translate(Concurrent($\alpha, \beta$)).

1. $PSM_\alpha \leftarrow translate(\alpha)$
2. $PSM_\beta \leftarrow translate(\beta)$
3. $PSM_{\alpha.final} \leftarrow CreateFinalStates(PSM_\alpha)$
4. $PSM_{\beta.inout} \leftarrow CreateExecutionClosure(PSM_\beta)$
5. $transitions_\alpha \leftarrow$ all transitions from $in(PSM_{\alpha.final})$
6. $C_\alpha^1 \leftarrow CompoundState(name_1, states(PSM_{\alpha.inout}))$
7. $C_\alpha^1 \leftarrow CompoundState(name_1, states(PSM_{\alpha.inout}))$
8. $P_1 \leftarrow ParallelState(pname_1, \{C_\alpha^1, C_\beta^1\})$
9. $t_1^{completion} \leftarrow Transition("done.state.pname_1")$
10. $P_1 \leftarrow P_1 \downarrow t_1^{completion}$
11. repeat with $\alpha$ and $\beta$ interchanged
12. return $PSM$ container $PSM_{Concurrent}$

4.4 Disabling Operator

The Disabling operator has two subtrees $\alpha$ and $\beta$, where $\alpha$ is regularly executed but can be “completely interrupted” (Paternò et al., 2012) by $\beta$ at any time. The remaining tasks of $\alpha$ are not executed once $\beta$ becomes active. This also implies that $\alpha$ is never executed if (one of) the first task(s) of $\beta$ is selected for execution before the first task(s) of $\alpha$.

From the Disabling operator point of view, $\alpha$ can be regarded as a compound state containing a partial state machine $PSM_\alpha$. A partial state machine $PSM_\beta$ (more precise, the in set of $PSM_\beta$) which implements $\beta$ is an “interrupting” partial state machine, which exits (or bypasses) the compound state containing $PSM_\alpha$.  

Figure 5: $PSM_{Concurrent}$ for sub-CTT “Operate TV”. The translation rule creates two parallel states in order to avoid that the in set of the PSM becomes mutually exclusive.
and switches to another execution branch. To finish the execution of $\alpha$ without interruption, the compound state representing $\alpha$ is equipped with an internal final state, which is entered when $\alpha$ (i.e., $PSM_{\alpha}$) was successfully executed. Due to the corresponding completion transition, the compound state is automatically left when $\alpha$ ($PSM_{\alpha}$, respectively) was executed without interruption of $\beta$ ($PSM_{\beta}$ respectively).

The formal translation steps of the Disabling operator are shown by Algorithm 4. Again, we apply the algorithm to the example of Figure 1. Because Disabling is the top-most operator, applying the algorithm completely translates our CTT. $\alpha$ in this case is the subtree “Operate TV” and $\beta$ is the subtree “Standby”. As usual, $PSM_{\alpha}$ and $PSM_{\beta}$ are generated by applying translate to the corresponding sub-CTTs. Next, we create the compound state for $PSM_{\alpha}$, which requires an additional final state that is connected to the out set (i.e., the transitions $t_1^{\text{completion}}$ and $t_2^{\text{completion}}$) of $PSM_{\alpha}$. This results in a new partial state machine $PSM_{\alpha\text{final}}$.

Before wrapping $PSM_{\alpha\text{final}}$ into the compound state, the ingoing transitions “Press Chan+ Button”, “Press Chan- Button”, “Press Vol+ Button” and “Press Vol- Button” of in($PSM_{\alpha\text{final}}$) must be preserved for subsequent connections. Consequently, the transitions of both in($PSM_{\alpha\text{final}}$) and in($PSM_{\beta}$) are extracted and the transitions of in($PSM_{\beta}$) (“Press Standby Button”) are duplicated to provide Connectables that can bypass $PSM_{\beta}$ and its corresponding compound state completely.

As in set of the resulting container $PSM_{\text{Disabling}}$, the extracted transitions of $PSM_{\alpha}$ and the duplicated ingoing transitions of $PSM_{\beta}$ are returned. The out set consists of the completion transition $t_1^{\text{completion}}$ and the out set of $PSM_{\beta}$. Consequently, the translated Disabling operator of the TV example is shown in Figure 6. Note that the transitions belonging to $PSM_{\alpha}$ point into the compound state and not just to it.

With the root operator of the CTT translated, we obtained a complete (yet partial) state machine from the CTT. However, this CTT is not executable yet, because it lacks proper initial and final states. By adding a common initial state, which is connected to the transitions of the $PSM_{\text{Disabling}}$’s in set and a common final state, which is connected with $PSM_{\text{Disabling}}$’s out set, we obtain a state machine which can be executed by standard state machine frameworks. We call this final step the creation of the “execution closure”.

---

**Algorithm 4: translate(Disabling($\alpha$, $\beta$)).**

```plaintext
PSM_{\alpha} \leftarrow \text{translate}(\alpha)
PSM_{\beta} \leftarrow \text{translate}(\beta)
PSM_{\alpha\text{final}} \leftarrow \text{CreateFinalStates}(PSM_{\alpha})
transitions_{\alpha} \leftarrow \text{all transitions from in}(PSM_{\alpha\text{final}})
transitions_{\beta} \leftarrow \text{all transitions from in}(PSM_{\beta})
transitions_{\beta}^{\text{dup}} \leftarrow \text{copy of transitions}_{\beta} \text{ with target}
C \leftarrow \text{CompoundState}(\text{cname}, \text{states}(PSM_{\alpha\text{final}}))
\text{intermediate} \leftarrow \emptyset
\text{for all } i \in \text{in}(PSM_{\beta}) \text{ do}
  C \leftarrow \text{connect } i
  \text{if state was created then}
    \text{intermediate} \leftarrow \text{intermediate } \cup \{\text{state}\}
  \text{end if}
\text{end for}
t_{\text{completion}} \leftarrow \text{Transition("done.state.cname")}
C \leftarrow \text{connect } t_{\text{completion}}
\text{intermediate} \leftarrow \text{intermediate } \cup \{t_{\text{completion}}\}
\text{return } \text{PSM container } PSM_{\text{Disabling}}
```
5 RELATED WORK

Task models have been found to be widely adapted in the research community. Several related work covers the translation of task models into corresponding UML diagrams, but in a limited way. For example, (Luyten and Clerckx, 2003; Van Den Bergh and Coninx, 2007) define mappings from CTTs to state machines, but neither provide a formal set of rules nor do they cover the full recursive case. Other work maps CTTs to UML activity diagrams in a similar way as in this work (Brüning et al., 2008; Brüning et al., 2012). However, we feel that state charts are easier to handle and use in actual implementations. Furthermore, (Zhu et al., 2012) extract state machines from CTTs to derive the system behavior. Unfortunately, they only use it for verification, but do not consider interactive execution. A recursive algorithm to map CTTs to state machines is presented in (Sinnig et al., 2013), but this only aims at extracting possible executions for verifications, and also does not consider interactive execution. Compared to this work, related projects that employ (CTT) task models to derive system information mostly seem to concentrate on the user interface (which is not enough when focusing a more comprehensive approach to model-driven systems engineering) or they generate UML models on a higher and more abstract level, which require some manual effort to implement.

6 CONCLUSION

In this work, we have presented the first recursive algorithm to translate CTTs, based on their tree structure, into executable state machines. Compared to prior work, the generated state machines interact with the environment by means of high-level events which correspond to tasks in CTTs. These events can be triggered by actual UI elements or concrete (sensor) events in real implementations. A full tool chain for this is presented in (Wagner, 2015). As the major technical contribution of this paper, we have introduced and extensively used incomplete or partial state machines. These are composed in a recursive way to form bigger and more complex (partial) state machines, which can act as central controllers for desktop or embedded applications.

REFERENCES